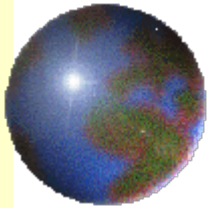
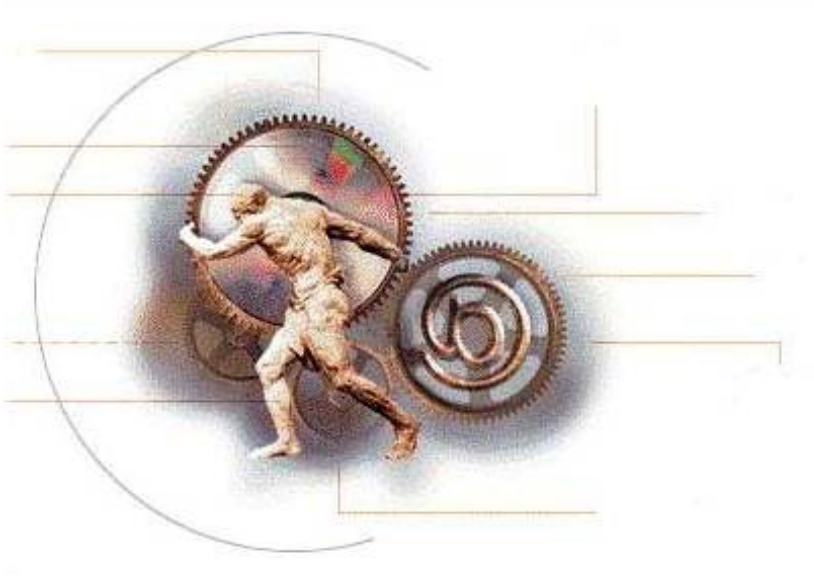


資料結構(Data Structures)

Course 0: 課程簡介



授課教師:陳士杰

國立聯合大學　資訊管理學系

Outline

- 了解本課程授課重點、目標及課程設計
- 了解本課程評分標準
- 課前重點

教材

- Text Book
  - 自製講義
  - 講義下載處: http://web.nuu.edu.tw/~sjchen (含上課錄影)
- Reference Book
  - Ellis Horowitz, Fundamentals of Data Structures in C/C++, 2nd, Computer Science Pre, 2006.
  - 徐熊健，資料結構與演算法，第二版，旗標出版社，2007.
  - 吳勁樺, 資料結構教學範本—使用 C++, 金禾資訊, 2004.
  - 洪逸, 資料結構(含精選試題), 鼎茂圖書.

課程重點

- Algorithm, Time Complexity (演算法與時間複雜度)
- Fundamentals of Recursive Algorithm (遞迴演算法基礎)
- Array (陣列)
- Link list (鏈結串列)
- Stack and Queue (堆疊與佇列)
- Tree and Binary Tree (樹與二元樹)
- Graph (圖形)
- Advanced Tree (高等樹)
- Search and Sort (搜尋與排序)
- Hashing (雜湊)

評分標準

- 期中考:45%
- 期末考:45%
- 平時成績:10%
- 程式加分作業:10%~15%

資料結構 v.s. 演算法

針對不同的問題，設計“資料在電腦中的**儲存方法**”

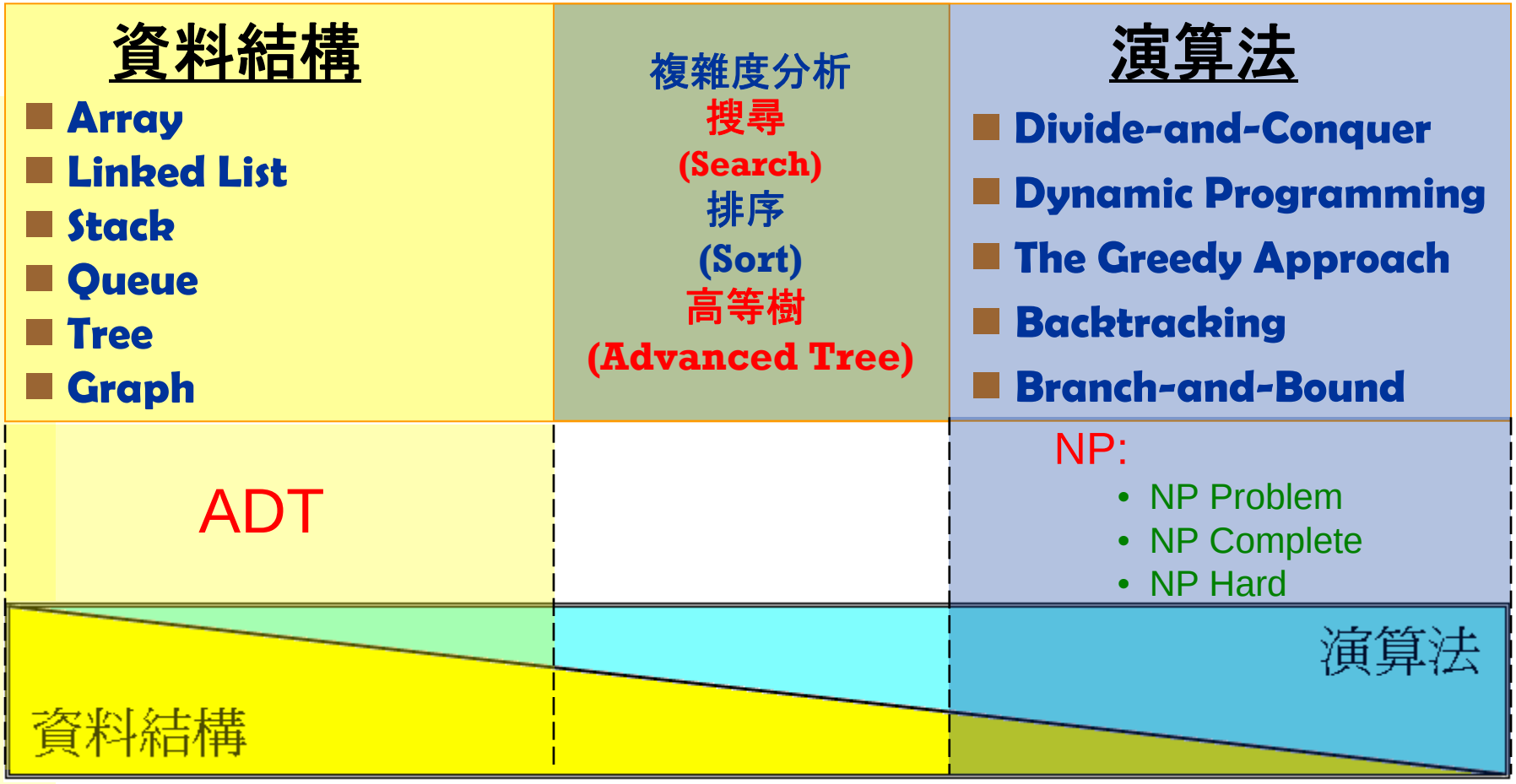
針對不同的問題，設計“**問題的解決步驟**”

程式 = 資料結構 + 演算法

- 使用最適當的【資料結構】，才能夠設計出最有效率的【演算法】，進而轉換成為有效率的【程式】。

- 從程式發展的複雜程度來看：
  - 程式簡單時，個別的資料可分別來處理!!
    - 如：int, char, double 等。
  - 但隨著程式的複雜，
    - 具有相關性的資料不應再被個別看待，最好將它們聚集起來。
    - 例如：學生的資料包括：學號、姓名、住址、電話、好幾個分數...等等。
    - 有意義的資料應被視為一個個體，如此可簡化程式的分析與設計。
- 程式的發展(develop)除了資料(data)，還有處理資料的步驟或方法，我們稱這是**演算法** (algorithm)。

資料結構 v.s. 演算法


資料結構
Array
Linked List
Stack
Queue
Tree
Graph
複雜度分析
搜尋
(Search)
排序
(Sort)
高等樹
(Advanced Tree)
演算法
Divide-and-Conquer
Dynamic Programming
The Greedy Approach
Backtracking
Branch-and-Bound
ADT
NP:
NP Problem
NP Complete
NP Hard
資料結構
演算法

- 為何要修資料結構與演算法？
  - 是程式發展的指導原則 (會寫程式不代表能寫出好的程式)
    - **原則** — 供作參考，不一定要遵循。但這些原則是前人的結晶，遵循著做，會比較省力
  - 可提昇程式發展能力
  - 可訓練解決問題時的邏輯思考能力
  - 資訊相關 (研究所、高考) 的考試會考

函式(function)

- 在設計結構化程式時，我們會將一個較大的程式切割成許多**子功能 (子程式)**，每個子程式或許可以再細分成數個更小的子功能。
- 我們最後再將這些子功能寫成**獨立的函式**。
- 我們也會將程式中**常出現**或**重複使用的程式區塊**獨立成一個函式。
- 完成函式的撰寫後，當主程式需要執行某一個函式時，就可以**直接叫用 (呼叫; call)** 所需的函式。

主程式

```
void main(void)
{
........
function1( ) ;
........
........
function2( ) ;
.....
}
```

函式呼叫
(Function Call)

函式 1

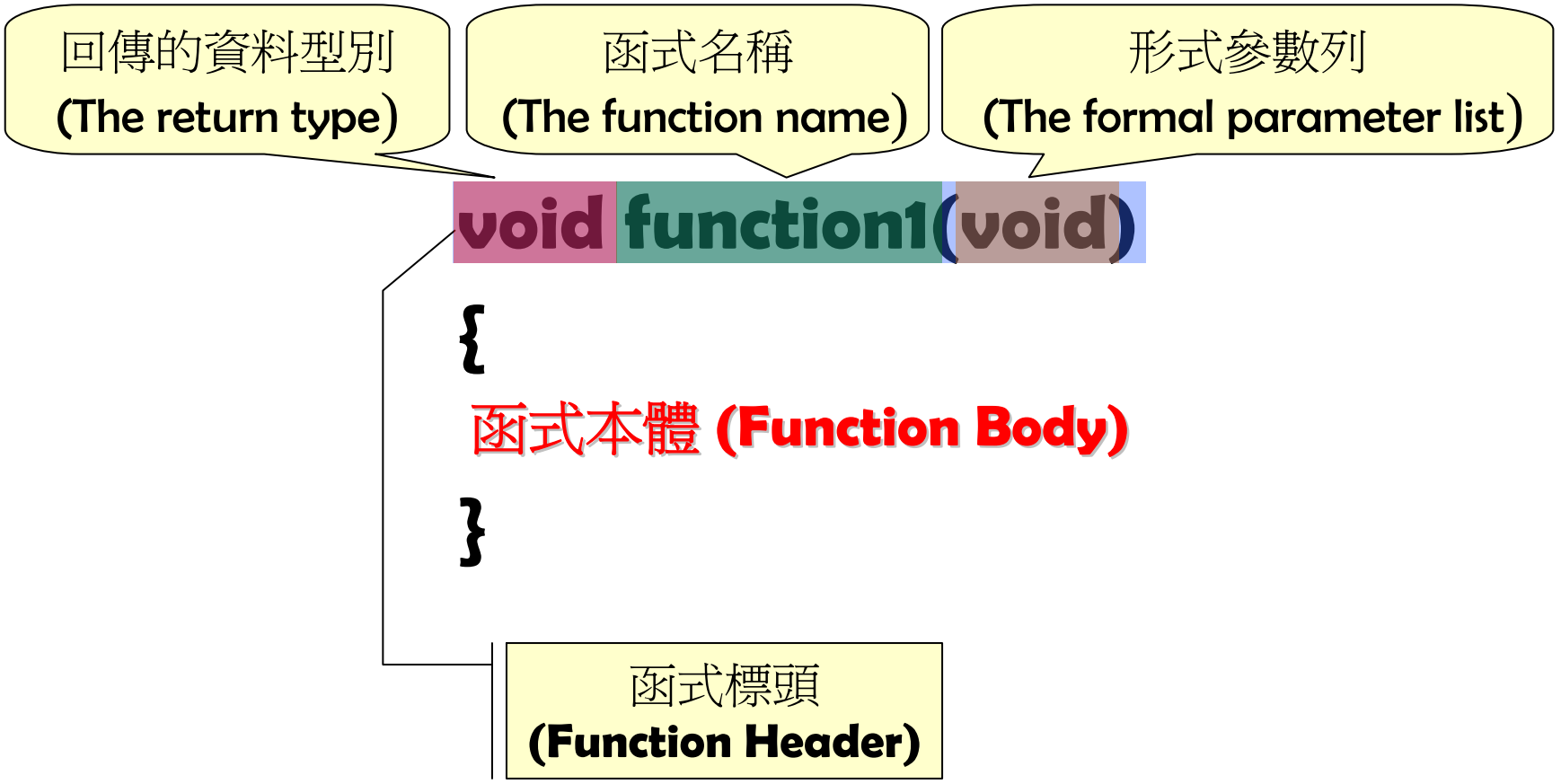
```
void function1(void)
{
........
}
```

函式 2

```
void function2(void)
{
........
function3( );
........
}
```

函式 3

```
void function3(void)
{
........
}
```


回傳的資料型別
(The return type)
函式名稱
(The function name)
形式參數列
(The formal parameter list)
void function1(void)
{
函式本體 (Function Body)
}
函式標頭
(Function Header)

參數傳遞 (Parameter Passing)

- **Call by address (傳址呼叫; Call by reference)**
  - 主函式呼叫子函式時，將**參數的位址**傳給子函式
  - 主、子函式佔用**相同的記憶體位置**
  - 因為主、子函式佔用相同的記憶體位置，故**繫結 (Binding) 最快**
  - 子函式執行過程中若改變參數值，則主函式的參數值也會改變 (即：**邊際效應 Side Effect**)
  - **Binding (繫結)**：
    - 決定程式執行的起始位址。
    - 通常，當程式所在的記憶體位址決定了，程式內的資料或變數被定義在記憶體的哪個位址也就確定了。

Call by value (傳值呼叫)

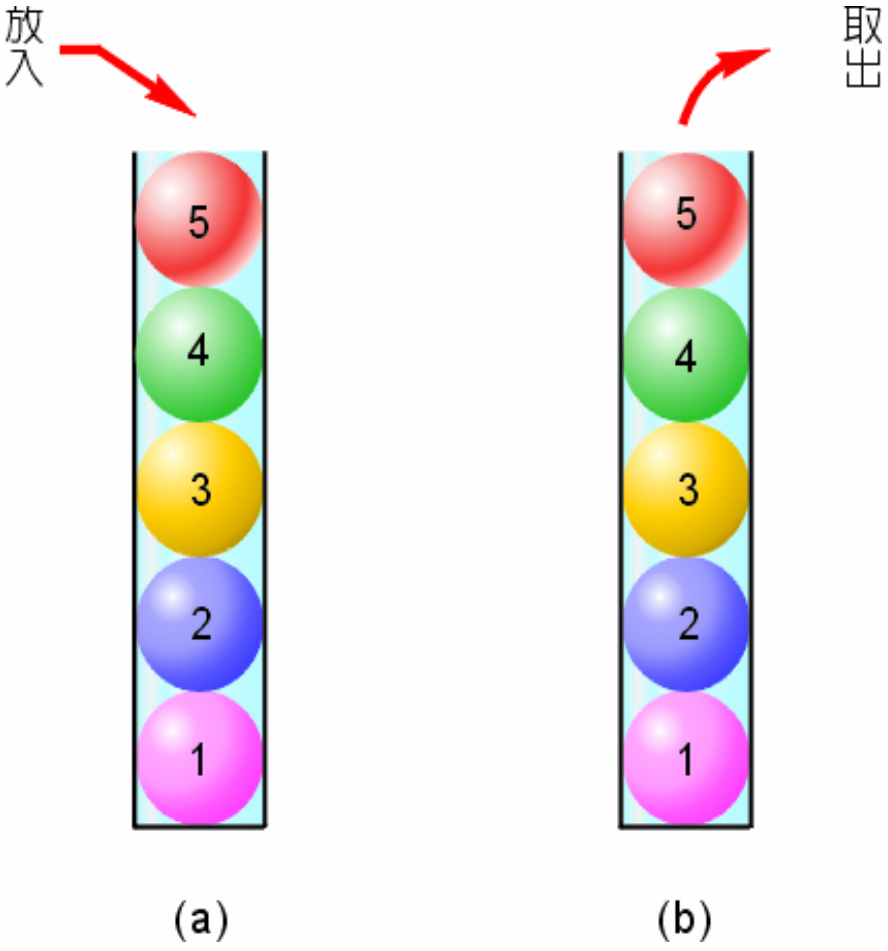
- 主函式呼叫子函式時，將主函式的值傳給子函式
- 作業系統**分派額外的記憶體位置**給子函式的參數值
- 因需要分派額外的記憶體位置，故**繫結 (Binding) 最慢**
- 子函式執行完畢，主函式的參數值不變 (即：**沒有Side Effect**)

Call by name (傳名呼叫)

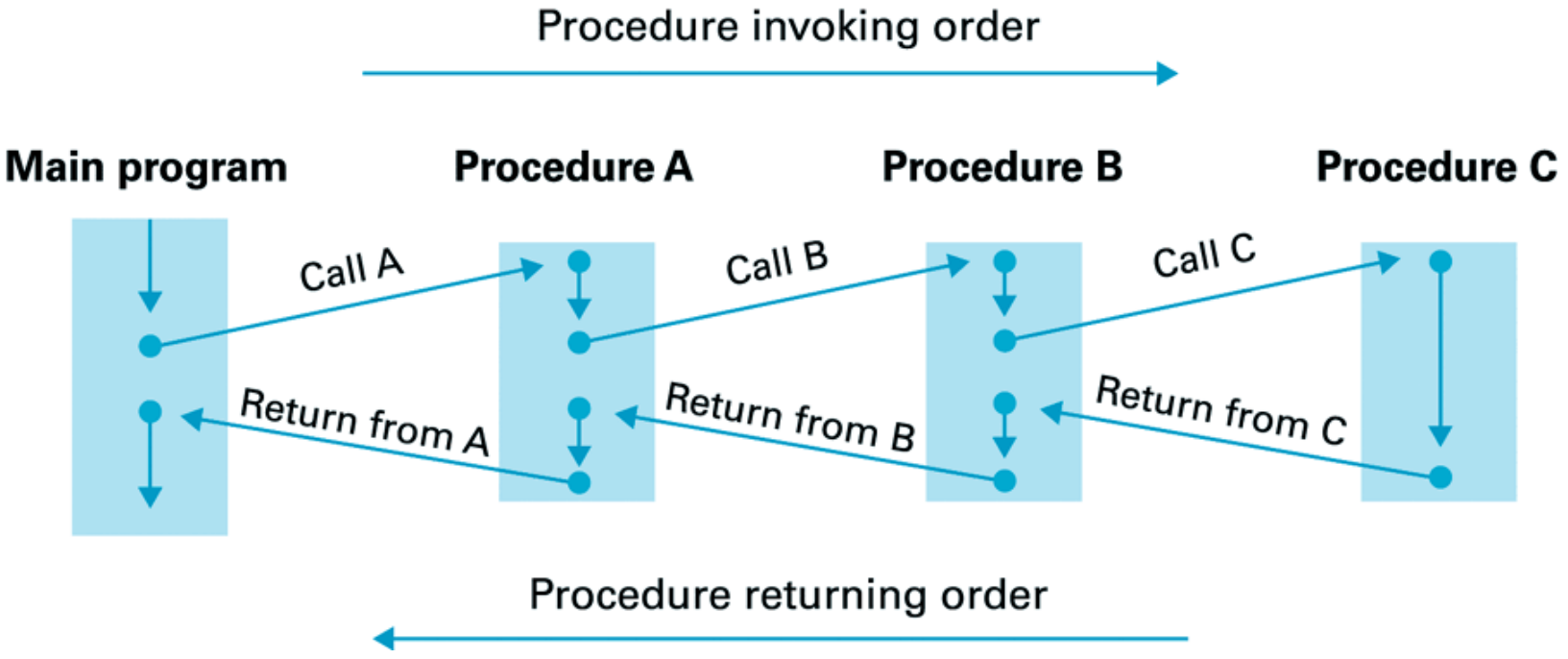
- 主函式呼叫子函式時，將主函式參數的名稱傳給子函式，並取代整個子函式所有對應之參數名稱
- 主、子函式**佔用相同的記憶體位置**
- 主、子函式雖然佔用相同的記憶體位置，但仍須改變所有子函式相對應的名稱，故**繫結 (Binding) 比Call by value 快，但是比Call by reference慢**。
- 子函式執行過程中若改變參數值，則主函式的參數值也會改變 (即：**有Side Effect**) 。

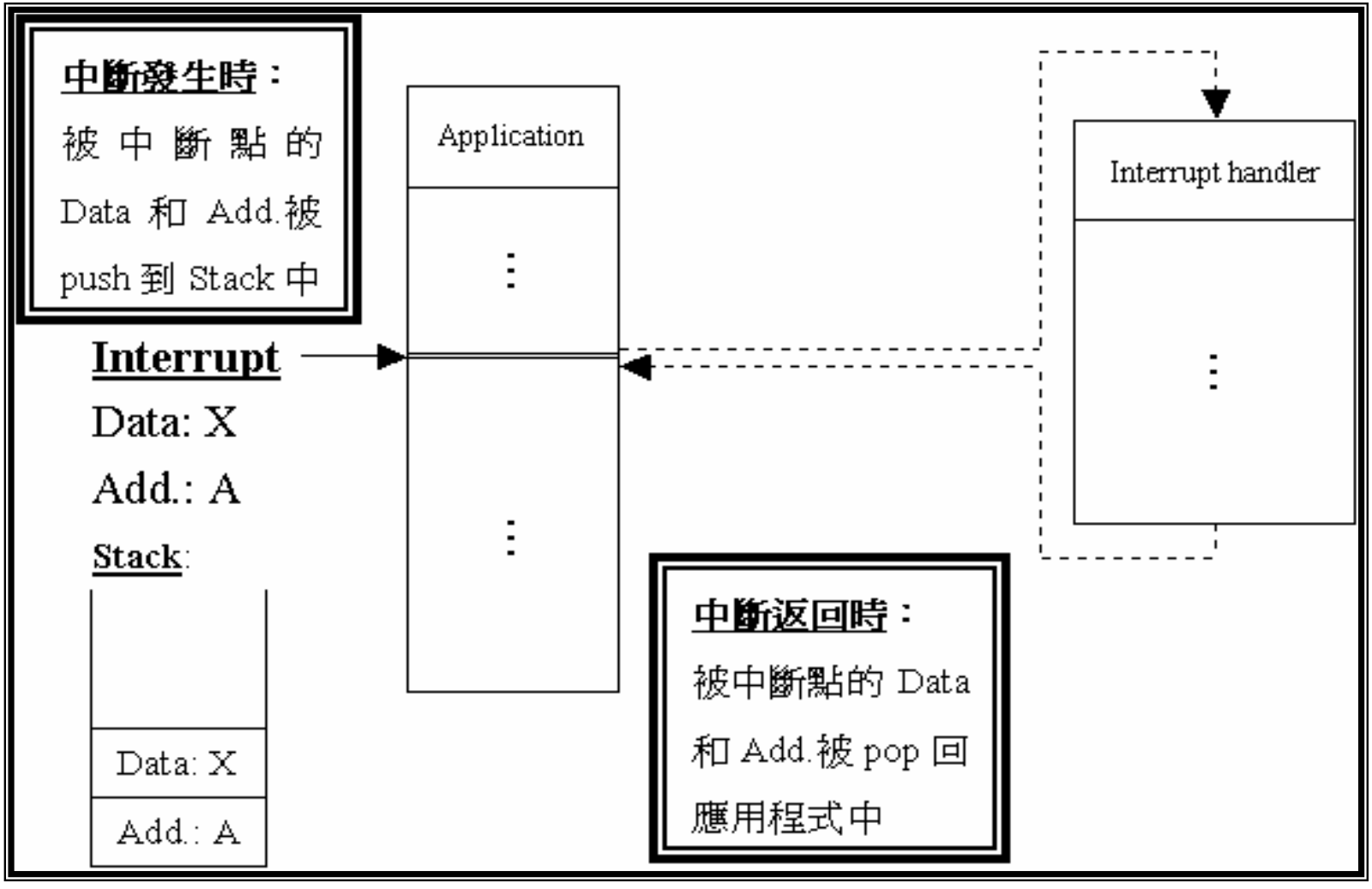
堆疊 (Stacks)

- 堆疊
  - **後進先出 (Last in, First Out)**
  - **先進後出 (First in, Last Out)**
- 右圖範例
  - 最早放進去的1號球會在球桶的最下方，而最後放進去的5號球會在球桶的最上方。
  - 要用球時，首先拿到的是球桶最上方的5號球，最後才會拿到1號球。



堆疊應用之處


Procedure invoking order
Main program
Procedure A
Procedure B
Procedure C
Call A
Call B
Call C
Return from A
Return from B
Return from C
Procedure returning order


中斷發生時：
被中斷點的 Data 和 Add.被 push 到 Stack 中
Application
Interrupt handler
Interrupt
Data: X
Add.: A
Stack:
Data: X
Add.: A
中斷返回時：
被中斷點的 Data 和 Add.被 pop 回應用程式中

遞迴(Recursion)的觀念

- Def: 在程式中含有自我呼叫的敘述存在
- **直接遞迴 (Direct Recursion):**
  - 函式或程序直接呼叫本身時稱之。

```
void Function2(void)
{
......
Function2( );
......
}
```

- **間接遞迴 (Indirect Recursion):**
  - 函式或程序先呼叫另外的函式，再從另外函式呼叫原來的函式稱之。

```
void Function1(void)
{
......
Function2( );
......
}
```

```
void Function2(void)
{
......
Function1( );
......
}
```